1 Overview

In this assignment you will learn the basics of Software Defined Networking (SDN) and a few of its applications. We will be working with OpenFlow, a communications interface defined between the control and forwarding layers of an SDN architecture. You will first use a simulator called Mininet to simulate a network topology that uses OpenFlow switches. Then, you will write the control logic for those switches to manipulate packets.

1.1 Command Line Interfaces

We will be working with various command line interfaces:

1. Commands prefixed with `$` are to be run in your VM terminal:

   ```
   $ command
   ```

2. Commands prefixed with `mininet>` are to be run via the Mininet prompt:

   ```
   mininet> command
   ```

   You can enter the Mininet prompt by running `sudo mn` in your VM terminal.

3. Commands prefixed with `h1>` are to be run in host `h1`'s console:

   ```
   h1> command
   ```

   You can enter, for example, host `h1`'s console by running `xterm h1` at the Mininet prompt. Note that the above applies to `h2`, `h3`, etc.

1.2 Python Resources

Students comfortable with Python may skip this section. The following resources may be helpful in familiarizing yourself with Python's syntax.

1. http://www.codecademy.com/en/tracks/python
2. https://docs.python.org/3/tutorial

2 VM Setup and Testing

In this section, you will set up a virtual machine (VM) that runs Mininet and OpenFlow. To get an idea of what is going on under the hood with Mininet, you may want to consult these two videos:

1. https://www.youtube.com/watch?v=cuej8DaQ0wk
2. https://www.youtube.com/watch?v=tn1-Pxm0ckc

Note: Mininet was still based on Python 2 when these videos were published.

2.1 Install and Configure VM Environment

Follow the instructions in `CS1430 PA1 VM Instructions.pdf`, which can be found on Canvas or at:
https://docs.google.com/document/d/1YMNCfLPM4scre4ZasjDBv6XUDtLbTTpHiVTFi6Dy0tM

2.2 Ping Harvard's Web Server!

With the VM running, SSH into your VM and then verify that it can access the Internet by pinging a known web server: `www.harvard.edu`. The following commands will allow you to easily launch additional terminals as well as ping the server.

```
ssh -X mininet@[Your VM IP here] # run this locally
$ xterm &                        # & detaches a process from the terminal
$ ping -c3 www.harvard.edu
```

[5 pts] Problem 1

What did you see? How many ICMP packets (ping requests) were sent out and replied to successfully by the web server? How long did the replies for each individual ping request take?

Solution: I see 3 ICMP packets and 3 were successfully received by the server. According the time is displayed on the right.
PING harvard-edu.go-vip.net (192.0.66.20) 56(84) bytes of data.
64 bytes from 192.0.66.20 (192.0.66.20): icmpseq=1 ttl=57 time=8.54 ms
64 bytes from 192.0.66.20 (192.0.66.20): icmpseq=2 ttl=57 time=8.47 ms
64 bytes from 192.0.66.20 (192.0.66.20): icmpseq=3 ttl=57 time=8.67 ms

3 Getting Started with Mininet

Mininet is a network emulation platform that creates a virtual OpenFlow network—controller, switches (or routers), hosts, and links—on a single virtual machine. Take a look at the following tutorial first: https://github.com/mininet/openflow-tutorial/wiki/Learn-Development-Tools

3.1 Deciphering the Network Topology

First, exit the Mininet prompt, making sure to clean up and clear the configuration using the `mn -c` command. As a general rule, you will always want to run a cleanup before deploying any topology. Next, start a new Mininet session with the default topology. The default topology consists of a switch (s1) and two connected hosts (h1, h2) as well as a reference OpenFlow Controller.

```
mininet> exit # exit prompt
$ sudo mn -c  # perform cleanup
$ sudo mn     # relaunch Mininet with default
mininet> h1 ping -c3 h2
mininet> h2 ping -c3 h1
```

[5 pts] Problem 2

Draw a precise diagram of the default network topology. Label each node and its network interface(s) correctly with IP addresses as in the figure from: https://github.com/mininet/openflow-tutorial/wiki/Learn-Development-Tools#start-network

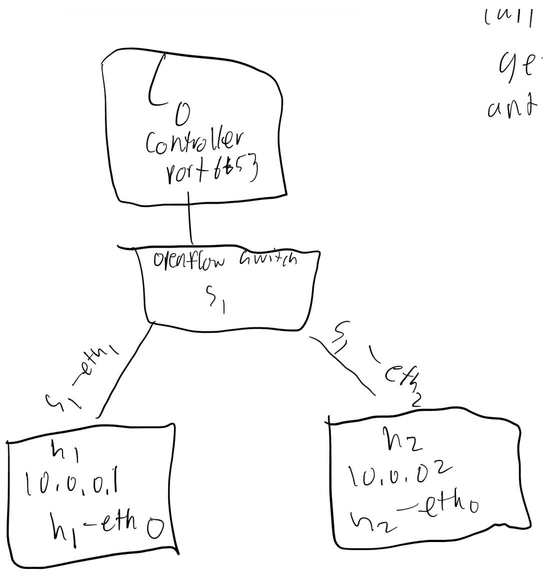


Figure 1: Diagram for problem 2

4 Something Finnicky with OpenFlow

Now we will use a new topology. Run the commands below and think about what the network topology looks like. You may want to try pinging a few hosts like before.

```
mininet> exit
$ sudo mn -c
$ sudo mn --topo linear,7 --mac --switch ovsk --controller remote
mininet> h1 ping -c3 h7
```

[0.5 pts] Problem 3a

Does pinging between hosts work? **No.**

[4.5 pts] Problem 3b

Explain the difference between this topology and the reference one from Problem 2. If you were able to ping successfully, draw a diagram of the network topology. If you were unable to ping successfully, explain how you could fix the issue.

Solution: So as the commands describe this one is a linear topology where the hosts are separated by a linear sequence of switches, where each host are connected their connected switch. In our current topology we have a remote controller the switches use to handle packets. However, the switches won't know how to forward packets beyond their direct connections, which is why the ping fails in our topology. So when I run "dpctl dump-flows" I see that the flow tables are empty so they don't know where to send their traffic for the pings to be reached by any host. This could be fixed by having a controller that is has the flow table for this topology since it is remote and the switches rely on it, or that the switches are given flows so they can correctly send packets to their destination.

4.1 Another Topology!

Let's look at another type of topology. Run the commands below and then answer the question that follows. This Mininet walkthrough might be helpful: http://mininet.org/walkthrough/

```
mininet> exit
$ sudo mn -c
$ sudo mn --topo tree,depth=2,fanout=2 --mac --switch ovsk
```

[5 pts] Problem 4

Draw a precise diagram of this network topology, labelling each node and its network interface(s) correctly with IP addresses (with same guidelines as before).

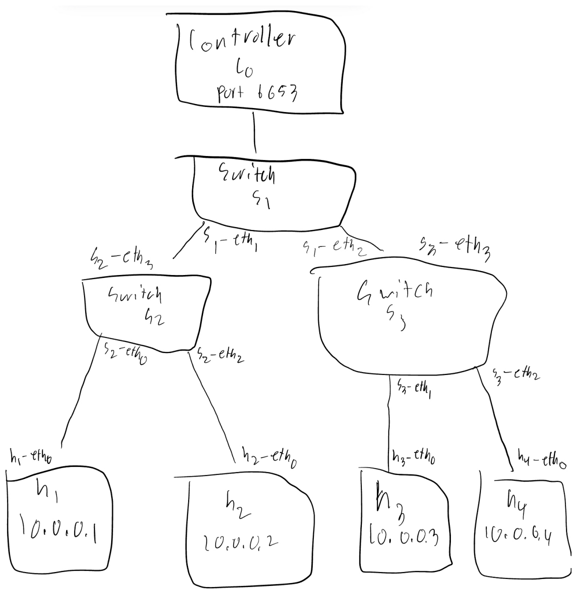


Figure 2: Network topology for Problem 4

5 Getting Started with POX

In this section, you will use an OpenFlow controller called POX. You will also learn how to write network applications (e.g., hubs and Layer-2 MAC learning) on POX and run them on a Mininet-based virtual network. POX is a Python-based SDN controller platform geared towards research and education. For more details, see: https://noxrepo.github.io/pox-doc/html/

5.1 Running a Simple Network

The network you will use in this section includes three hosts and a switch with an OpenFlow controller (POX):

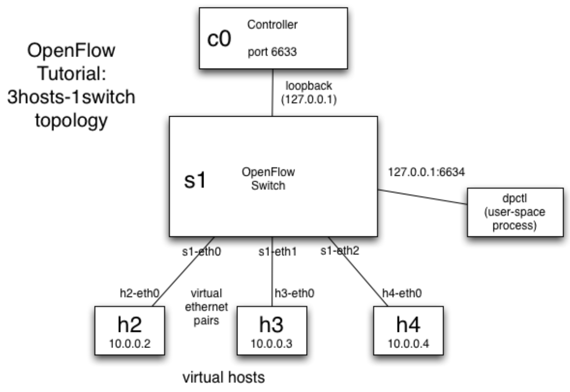


Figure 3: Simple network with hosts and a controller.

Let's get started by cleaning up Mininet and making sure there are no controllers already running on the system. This is generally good practice, so if you have any bugs or issues, always make sure there are no zombie processes left alive first. Run the following code:

```
mininet> exit
$ sudo fuser -k 6633/tcp
$ sudo mn -c
```

Next, we will start both a Mininet virtual network and a POX controller. You will want to have two terminals for your VM open simultaneously. To do this, simply open another local terminal and SSH in as usual. First, let's start up our Mininet virtual network:

```
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

At this point, the first terminal should show the `mininet>` prompt. Note that we have not yet started a controller! When an OpenFlow switch loses connection to a controller (or starts without one), it will attempt to reestablish contact with the controller, increasing the delay between connection attempts, up to a maximum of 15 seconds.

Next, in your second terminal, let's start a controller for our network to connect to. Mininet comes with a few basic controllers in the `pox` directory. Let's use a basic hub controller that functions by sending all incoming packets out of each of its ports (i.e., to every device it is connected to). Run the following commands:

```
$ ~/pox/pox.py log.level --DEBUG forwarding.hub
```

Starting the hub in a verbose debug mode outputs logging information to the terminal, allowing us to see when a switch connects. At this point, the switches may take some time to connect, generally anywhere between 0 and 15 seconds, as they started without a controller. If this is too long, the switch may be configured to wait no more than N seconds using the `--max-backoff` parameter in the future. For now, you can exit Mininet to remove the switch(es) and then restart Mininet to immediately connect to our already active controller. Wait until the application indicates that the OpenFlow switch has connected. POX will print a message similar to this:

```
INFO:openflow.of_01:[Con 1/1] Connected to 00-00-00-00-01
DEBUG:samples.of_tutorial:Controlling [Con 1/1]
```

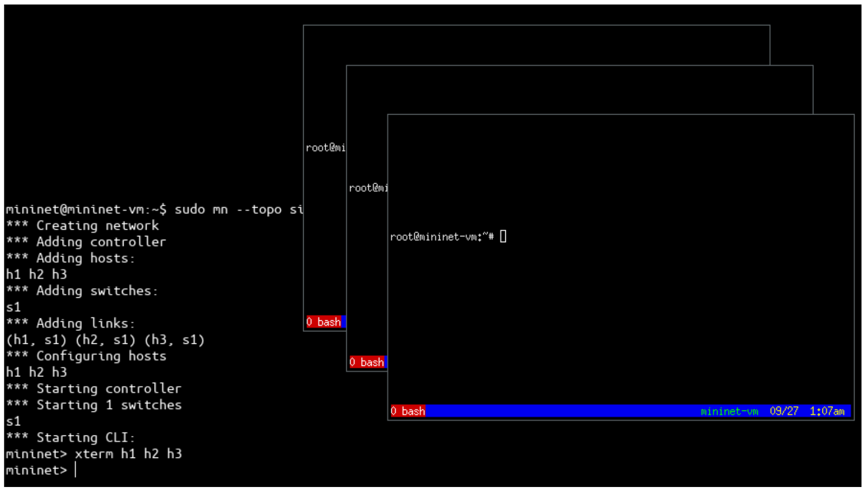
Note: to avoid delays, you can start the POX controller *before* initializing the Mininet network.

5.2 Testing the Hub Controller

Now, verify all hosts can ping each other and see the exact same traffic—the expected behavior of a hub. We will need to be able to run command to control and view the traffic in each host. In the Mininet console, run the following to start up the separate xterms (console windows):

```
mininet> xterm h1 h2 h3
```

Arrange the terminals so you can see all of them simultaneously. It may look something like this:



```
mininet@mininet-vm:~$ sudo mn --topo si
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Starting controller
*** Starting 1 switches
s1
*** Starting CLI:
mininet> xterm h1 h2 h3
mininet> |
```

```
root@mi
```

```
root@mi
```

```
root@mininet-vm:~# []
```

Figure 4: Mininet prompt with multiple xterm windows open.

To inspect traffic, we will use a utility called `tcpdump`, which prints the packets a host receives on a specified network interface. Run the following commands *in order* (double-check the prefixes and take note of which host each command is run onr):

```
h2> tcpdump -XX -n -i h2-eth0 # run tcpdump on h2
h3> tcpdump -XX -n -i h3-eth0 # run tcpdump on h3
h1> ping -c1 10.0.0.2         # ping from h1
```

[2.5 pts] Problem 5a

You should see two types of packets in the `tcpdump` output on the xterms of h2 and h3. What are the two types? Draw the path of ping packets on a diagram of the network (both request and response packets).

Solution: There appears to be ARP and ICMP requests and replies (responses).

[2.5 pts] Problem 5b

Now, see what happens when a non-existent host fails to reply (e.g., `ping -c1 10.0.0.5`). How many packets and of what type do you see in the `tcpdump` output for h2 and h3?

Solution: There were 3 packets trasmitted and they were all ARP requests, of course asking for the Mac address of 10.0.0.5, but there was no reply obviously because this host didn't exist. So h1 tried to find 10.0.0.5 3 times.

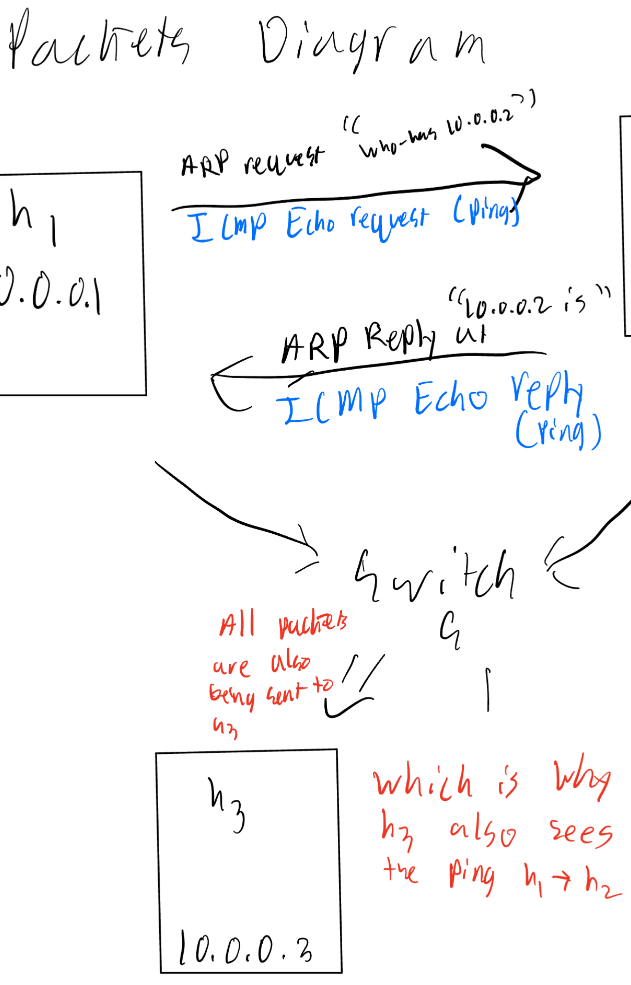

Packets Diagram
h1
10.0.0.1
h2
10.0.0.2
ARP request "who-has 10.0.0.2"
ICMP Echo request (ping)
ARP Reply "10.0.0.2 is" at
ICMP Echo reply (ping)
Switch
S1
All packets are also being sent to h3
h3
10.0.0.3
which is why h3 also sees the ping h1 → h2

Figure 5: Diagram for 5a

5.3 Benchmarking the Hub

Now we will perform some benchmarking of the network topology and hub controller. For now, we simply want to test how much data a hub can handle. Luckily, there is a nice tool called `iperf` that can help us. Run the following commands on the console with the Mininet prompt:

```
mininet> pingall  # verify all nodes are reachable
mininet> iperf    # perform the benchmark
```

[2.5 pts] Problem 5c

What sort of speeds are you seeping? The virtual links in Mininet have a very high bandwidth, so they should not be a bottleneck/reason for slow speeds. Based on what you now know about OpenFlow and this hub, can you give a few reasons for why speeds may be slower than expected?

Solution: To eventually answer

```
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h3
.*** Results: ['92.1 Gbits/sec', '92.3 Gbits/sec']
```

Quick Note: Hub Code

You can look at the hub code provided by Mininet and POX at `~/pox/pox/forwarding/hub.py`. This will be useful for the next few sections.

6 Switch Controller

This time, let's verify that hosts can ping each other when the controller is behaving like a Layer-2 learning switch. Kill the POX controller by pressing Ctrl-C (or Cmd-C) in the terminal running the controller program. We can launch the L2 learning switch and utilize the 3 xterms from before. To set things up, first start the controller and network in separate windows:

```
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
$ pox.py log.level --DEBUG forwarding.l2_learning # starting the learning switch!
```

Launch xterms for the Mininet hosts and run the commands as we did previously:

```
mininet> xterm h1 h2 h3
h2> tcpdump -XX -n -i h2-eth0
h3> tcpdump -XX -n -i h3-eth0
h1> ping -c1 10.0.0.2
```

[5 pts] Problem 6

What do you see? Examine the code of the L2 learning controller at `~/pox/pox/forwarding/l2_learning.py`. Explain the behavior of this controller.

Solution: So using the controller before, the host would always sent an ARP packet prior to pinging the host everytime. However, after the first ping, when I send multiple pings there are no more ARP packets sent. This is what I observed in the tcpdump terminal.

The L2 learning controller is like a learning switch that "learns" as it processes packets. It maintains a SELF.MACTOPORT table to track which devices connect to which ports. So when a packet arrives, the controller records the source MAC address and incoming port, then checks if it knows where the destination is located. For unknown destinations, it sends the packet to all ports, but for known destinations, it installs a specific flow rule with timeouts in the switch to direct traffic only to the destination port. This explains why ARP broadcasts are only needed for the first ping and hwy after the controller learns the location of hosts, it creates direct forwarding paths between them.

Additional Material

The POX Wiki at `https://noxrepo.github.io/pox-doc/html/` can give you a complete tour—skim the wiki if you have any questions.

7 Programming Custom Network Topology

In this section, you will learn how to build a custom topology using the Mininet Python API and how parameters such as bandwidth, delay, loss, and queue size can be set individually for different links in the topology. You will also learn how to test the performance of a topology using `ping` and `iperf`. After the overview, you will create your own custom topology based on a common 3-tier datacenter architecture (i.e., core, aggregation, and edge). Make sure to follow each step carefully.

7.1 Overview

The network you will use in this exercise connects hosts and switches in a linear topology as shown:

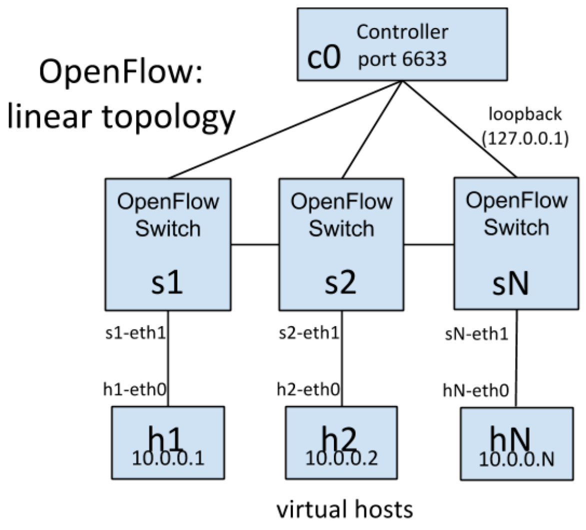


Figure 6: Linear network topology.

The code for this topology can be found in the assignment folder as `topologies/LinearTopo.py` as well as on the following page:

```python
#!/usr/bin/python
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.util import dumpNodeConnections
from mininet.log import setLogLevel


class LinearTopo(Topo):
    "Linear topology of k switches, with one host per switch."

    def __init__(self, k=2, **opts):
        """
        k: number of switches (and hosts)
        hconf: host configuration options
        lconf: link configuration options
        """
        super(LinearTopo, self).__init__(**opts)

        self.k = k
        lastSwitch = None
        for i in range(1, k):
            host = self.addHost("h%s" % i)
            switch = self.addSwitch("s%s" % i)
            self.addLink(host, switch)

            if lastSwitch:
                self.addLink(switch, lastSwitch)

            lastSwitch = switch


def simpleTest():
    "Create and test a simple network"
    topo = LinearTopo(k=4)
    net = Mininet(topo)
    net.start()
    print("Dumping host connections")
    dumpNodeConnections(net.hosts)
    print("Testing network connectivity")
    net.pingAll()
    net.stop()


if __name__ == "__main__":
    setLogLevel("info")  # Tell mininet to print useful information
    simpleTest()
```

The important classes, methods, functions, and variables in the above code include:

- `Topo`: the base class for Mininet topologies
- `addSwitch(name)`: add a switch to the topology and return the switch name
- `addHost(name)`: add a host to the topology and return the host name
- `addLink(a, b)`: add a bidirectional link to the topology (and return the link key)
  Note: Links in Mininet are bidirectional unless noted otherwise.
- `Mininet`: main class to create and manage a network
- `start()`: start the network
- `pingAll()`: test connectivity by having all nodes ping each other
- `stop()`: stop the network
- `net.hosts`: all hosts in the network
  Note: this is an instance attribute; in our case, `net` is the `Mininet` class instance.
- `setLogLevel(level)`: set Mininet's default output level
  Note: valid levels are: `'info' | 'debug' | 'output'`

7.2 Setting Performance Parameters

In addition to basic behavioral networking, Mininet provides performance limiting and isolation features through abstractions such as the `CPULimitedHost` and `TCLink` classes. The code for this topology can be found in the assignment folder as `topologies/LinearTopoPerf.py` as well as on the following page:

```python
#!/usr/bin/python
from mininet.link import TCLink
from mininet.log import setLogLevel
from mininet.net import Mininet
from mininet.node import CPULimitedHost
from mininet.topo import Topo
from mininet.util import dumpNodeConnections, irange


class LinearTopo(Topo):
    "Linear topology of k switches, with one host per switch."

    def __init__(self, k=2, **opts):
        """
        k: number of switches (and hosts)
        hconf: host configuration options
        lconf: link configuration options
        """
        super(LinearTopo, self).__init__(**opts)
        self.k = k
        lastSwitch = None
        for i in irange(1, k):
            host = self.addHost("h%s" % i, cpu=0.5 / k)
            switch = self.addSwitch("s%s" % i)

            # 10 Mbps, 5ms delay, 1% loss, 1000 packet queue
            self.addLink(host, switch, bw=10, delay="5ms", loss=1,
                         max_queue_size=1000, use_htb=True)
            if lastSwitch:
                self.addLink(switch, lastSwitch, bw=10, delay="5ms", loss=1,
                             max_queue_size=1000, use_htb=True)

            lastSwitch = switch


def perfTest():
    "Create network and run simple performance test"
    topo = LinearTopo(k=4)
    net = Mininet(topo=topo, host=CPULimitedHost, link=TCLink)
    net.start()
    print("Dumping host connections")
    dumpNodeConnections(net.hosts)
    print("Testing network connectivity")
    net.pingAll()
    print("Testing bandwidth between h1 and h4")
    h1, h4 = net.get("h1", "h4")
    net.iperf((h1, h4))
    net.stop()


if __name__ == "__main__":
    setLogLevel("info")
    perfTest()
```

7.3 Running the Custom Topologies

To run the custom topology above, download **LinearTopo.py**.

```
$ cd ~
$ chmod u+x LinearTopo.py
$ sudo ./LinearTopo.py
```

To run the topology with performance parameters, use **LinearTopoPerf.py** instead.

7.4 Programming a Datacenter Network Topology

Datacenter networks typically have a tree-like topology. End hosts connect to top-of-rack switches, which form the leaves (edges) of the tree. One or more core switches form the root, and one or more layers of aggregation switches form the middle of the tree. In a basic tree topology, each switch (except the core switch) has a single parent. Additional switches and links may be added to form more complex topologies (e.g., fat tree) to improve fault tolerance or increase total bandwidth.

In this section, your task will be to create a simple tree topology. You will assume each level (i.e. core, aggregation, edge, and host) to be composed of a single layer of switches and hosts with a configurable fanout value (k). For example. a simple tree network having a single layer per level with fanout $k = 2$ looks like:

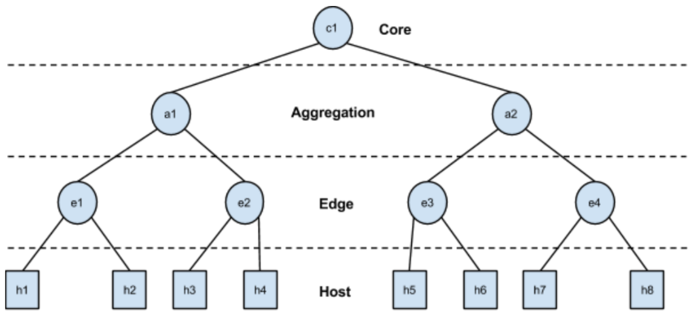


Figure 7: Datacenter tree topology.

To begin, download and unzip `q7.zip`. The file `CustomTopo.py` contains a template which takes the following arguments as input:

- `linkopts1`: performance parameters for links between core and aggregation switches
- `linkopts2`: performance parameters for links between aggregation and edge switches
- `linkopts3`: performance parameters for links between edge switches and hosts
- `fanout`: fanout degree (children per node)

[5 pts] Problem 7

Create the tree-based datacenter topology described above. Your logic should support setting at least the `bw` and `delay` parameters for each link.

7.5 Testing Your Code

Edit `CustomTopo.py` to pass in `linkopts1`, `linkopts2`, and `linkopts3` when the class is instantiated. You may uncomment the sample code and link options we provide to test the topology:

```
linkopts1 = dict(bw=10, delay="5ms", loss=10, max_queue_size=1000, use_htb=True)
linkopts2 = dict(bw=10, delay="5ms", loss=10, max_queue_size=1000, use_htb=True)
linkopts3 = dict(bw=10, delay="5ms", loss=10, max_queue_size=1000, use_htb=True)
topos = { "custom": ( lambda: CustomTopo(linkopts1,linkopts2,linkopts3) ) }
```

Run the Mininet ping test with the custom topology:

```
$ sudo mn --custom CustomTopo.py --topo custom --test pingall --link tc
```

If everything has been set up correctly, you should observe the behavior specified by the link options provided. Feel free to adjust the parameters shown above (e.g., `bw`, `delay`, `loss`) to see what effects they may have.

8 Layer 2 Firewall

In this section, you will create a network application that implements a Layer 2 firewall. This firewall selectively disables inbound and outbound traffic between two systems based on their MAC address.

8.1 Overview

A firewall is a network security system that is used to control the flow of ingress and egress traffic—usually between a more secure local-area network (LAN) and a less secure wide-area network (WAN). The system analyzes packets for parameters such as L2/L3 headers (i.e., MAC and IP) or performs deep packet inspection (DPI) for higher layers (e.g., application type, services, etc.) to filter network traffic.

POX allows running multiple applications concurrently (e.g., MAC learning in conjunction with firewalling), but it does not automatically handle rule conflicts. You have to ensure that conflicting rules are not being installed by multiple applications (e.g., both applications trying to install rules with the same source and destination MACs at the same priority level but with different actions). The most simplistic way to avoid this contention is to assign different priority levels to each application.

8.2 Programming a Firewall

In this section, your task is to implement a Layer 2 firewall that runs alongside the MAC learning module on the POX OpenFlow controller. The firewall will be provided a list of MAC address pairs in the form of an access control list (ACL). When a connection establishes between the controller and the switch, the application installs flow rule entries in the OpenFlow table to disable all communication between the specified MAC pairs. Your firewall should be agnostic of the underlying topology and should take a MAC pair list as input to install rules on the switches. To make things easier, we will simply install the rules on **all** switches in the network.
To get started, download and unzip `q8.zip`. You should see:

- `firewall.py`: a skeleton class you will update with logic for installing firewall rules.
- `firewall-policies.csv`: list of MAC pairs (policies) read as input by the firewall.

The policy list (`firewall-policies.csv`) specifies MAC addresses that are not permitted to communicate with each other. You will not need to modify the policy list. The skeleton class inside `firewall.py` contains a function named `_handle_ConnectionUp()`. The file also contains a global variable, `policyFile`, that should hold the path of `firewall-policies.csv`. Whenever a connection is established between the POX controller and the OpenFlow switch, `_handle_ConnectionUp()` is called.

[5 pts] Problem 8

Update the `_handle_ConnectionUp()` function to install rules for the OpenFlow switches so that they drop any incoming packet containing a source and destination MAC address pair forbidden by our policy list. (Note: make sure to handle any conflicts carefully—we will be testing with different policy lists!)

8.3 Testing Your Code

Copy `firewall.py` and `firewall-policies.csv` into `~/pox/pox/misc` on your VM. The rule shown below disables all communication between hosts h1 and h2:

```
id,mac_0,mac_1
1,00:00:00:00:00:01,00:00:00:00:00:02
```

With this rule in place, let's see what happens when we attempt to ping one host from the other. Take note that we are invoking the POX controller command with an ampersand suffix (`&`), which will start the process in the background and allow us to continue using the same terminal for additional commands. Alternatively, you may choose to run the POX controller and Mininet commands in separate terminals.

```
$ cd ~
$ sudo fuser -k 6633/tcp
$ pox.py forwarding.l2_learning misc.firewall &     # start POX controller
$ sudo mn --topo single,3 --controller remote --mac # deploy Mininet topology
mininet> h1 ping -c1 h2 # ping h2 from h1
```

If everything has been set up correctly, you should see that neither host h1 nor host h3 are able to successfully ping host h2. However, host h1 should still be able to ping host h3 (and vice versa) since there is no flow rule installed that forbids communication between the two.

9 Custom OpenFlow Controller - Dijkstra's Algorithm

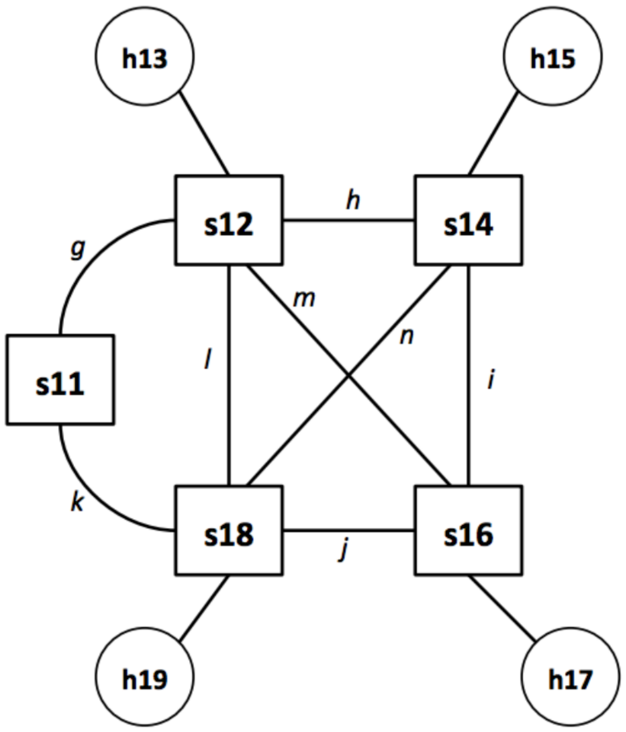
In this section, you will utilize Dijkstra's algorithm to find the shortest path between every pair of nodes. You will then write an OpenFlow controller that leverages the algorithm to dynamically create its own routing table.

9.1 Getting Started

To get started, download and unzip `q9.zip`. You will see the following files:

- `topo.py`: a skeleton class you will update with logic for creating the network topology with custom delays shown below
- `dijkstra.py`: a skeleton class you will update with shortest-path routing logic
- `delay.csv`: a list of link, delay value pairs to be read by the controller

We will use a specific topology and set of delays as depicted in Figure 8. The delay values in the table are from `delay.csv`.



Link	Delay(ms)
g	10
h	50
i	10
j	30
k	30
l	10
m	100
n	20

Figure 8: Custom topology and delay table

[5 pts] Problem 9a

Implement the topology depicted in Figure 8 in `topo.py`.

[5 pts] Problem 9b

Factoring in the delays shown for each link, manually determine the "shortest" paths (i.e., those with the least delay) for each pair of hosts in the topology. Which nodes are in each shortest path and how much time does each path require? Note: use Dijkstra's algorithm.

[5 pts] Problem 9c

Write pseudocode for your controller using Dijkstra's algorithm to find the shortest paths between all pairs of nodes given a topology and link delay table. You may assume that a dictionary mapping link labels to delay values exists (such as in `delay.csv`).

[5 pts] Problem 9d

Implement your code in `dijkstra.py`. Notice that the filename `delay.csv` is assigned to the variable `delayFile`. You may assume a static topology shape and hard-code link labels (e.g., g, h, i, etc.), but **do not** hard-code the delays: your implementation must be general enough to function properly even with different delay values.

9.2 Testing your Code

Once you have your code, copy `dijkstra.py` and `delay.csv` into the `~/pox/pox/misc` directory on your VM. Run the following commands to see your code in action:

```
$ cd ~/pox/pox/misc/
$ pox.py misc.dijkstra &
$ sudo mn --custom topo.py --topo custom --controller remote --mac --link tc
```

In Mininet, try pinging host (h17) from host (h13). If everything has been set up correctly, host (h13) will ping (h17), showing a RTT corresponding to the shortest path you earlier. You may wish to additionally ping between each pair of hosts to validate your results.

10 Submitting the Assignment

Your submission should be a `.zip` archive with a `CS1430_PA1_` prefix followed by each team member's full name separated with two underscores from the next name. Only one submission per team is needed. The archive should contain:

- PDF write-up
- Assignment code
- README file

Example filename: `CS1430_PA1_Jane_Doe__John_Doe__Jane_Smith__John-Paul_Smith.zip`

Write-up

Written responses (and any sketches, diagrams, and screenshots) should be contained within a single PDF document. (LaTeX is highly recommended!) Each response or figure should clearly indicate which problem is being answered. The write-up must contain the full names of all team members.

Code

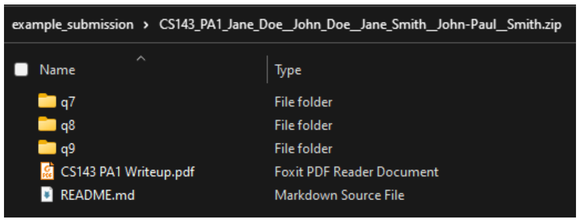
Code should be organized in directories indicating the question they are for. You should include **all** files that were provided, but with the changes you made. You may choose to follow our naming scheme (e.g., `q7/`) or use your own, but it should be apparent which files are for which problem.

README

The README should contain instructions on how to run code you submit (assuming we use the same starting VM image). Make sure to include details about: where files should be placed, what commands should be run, etc. The README must also contain the full names of all team members.

Example

An example submission might look like this:



example_submission › CS143_PA1_Jane_Doe__John_Doe__Jane_Smith__John-Paul__Smith.zip

Name	Type
q7	File folder
q8	File folder
q9	File folder
CS143 PA1 Writeup.pdf	Foxit PDF Reader Document
README.md	Markdown Source File

Figure 9: Example submission.

Congratulations!

Congratulations on completing your assignment!